CHAPTER 7

Python and MongoDB

Python is by far one of the easier programming languages to learn and master. It's an especially great language to start with if you are relatively new to programming. And you'll pick it up that much more quickly if you're already quite familiar with programming.

Python can be used to quickly develop an application while ensuring the code itself remains perfectly readable. With that in mind, this chapter will show you how to write simple yet elegant, clear, and powerful code that works with MongoDB through the Python driver (AKA, the PyMongo driver).

First, you'll look at the `Connection()` function, which enables you to establish a connection to the database. Second, you'll learn how to write documents, or *dictionaries*, as well as how to insert them. Third, you'll learn how to use either the `find()` or `find_one()` command to retrieve documents using the Python driver. Both of these commands optionally take a rich set of query modifiers to narrow down your search and make your query a little easier to implement. Fourth, you'll learn about the wide variety of operators that exist for performing updates. Finally, you'll take a look at how to use PyMongo to delete your data at the document or even the database level. As an added bonus, you'll learn how to use `DBRef` module to refer to data stored elsewhere.

Throughout the chapter, you'll see many practical code examples that illustrate the examples discussed. The code itself will be preceded with a greater than (>) symbol to indicate the command gets written in the Python shell. The query code will be styled in **bold**, whereas the resulting output will be rendered in plaintext. Let's get started.

Working with Documents in Python

As mentioned in earlier chapters, MongoDB uses BSON-styled documents, and PHP uses associative arrays. In a similar vein, Python has what it calls *dictionaries*. If you've already played around with the MongoDB console, we're confident you are absolutely going to *love* Python. After all, the syntax is so similar that the learning curve for the language syntax will be negligible.

We've already covered the structure of a MongoDB document in the preceding chapter, so we won't get into that again now. Instead, let's examine what a document looks like in the Python shell:

```
item = {
    "Type" : "Laptop",
    "ItemNumber" : "1234EXD",
    "Status" : "In use",
    "Location" : {
        "Department" : "Development",
    "Building" : "2B",
        "Floor" : 12,
        "Desk" : 120101,
        "Owner" : "Anderson, Thomas"
    },
    "Tags" : ["Laptop","Development","In Use"]
}
```

While you should keep the Python term *dictionary* in mind, in most cases this chapter will refer to its MongoDB equivalent, *document*. After all, most of the time, we will be working with MongoDB documents.

Using PyMongo Modules

The Python driver works with modules. You can treat these much as you treat the classes in the PHP driver. Each module within the PyMongo driver is responsible for a set of operations. There's an individual module for each of the following tasks (and quite a few more): establishing connections, working with databases, leveraging collections, manipulating the cursor, working with the `DBRef` module, converting the `Object ID`, and running server-side JavaScript code.

This chapter will walk you through the most basic yet useful set of operations needed to work with the PyMongo driver. Step-by-step, you learn how to use commands with simple and easy-to-understand pieces of code that you can copy and paste directly into your Python shell (or script). From there, it's a short step to managing your MongoDB database.

■ **Note** Commands will be styled in **bold** in code and have a prefix that is preceded by three greater than symbols (**>>>**). This convention indicates that the line introduces a new command that is typed into the shell. Code that starts with three dots (...) indicates that the code is continued from the preceding line. The resulting output will not be styled.

Connecting and Disconnecting

Establishing a connection to the database requires that you first import the PyMongo driver into Python itself. This is an absolute prerequisite; otherwise, none of the modules will be loaded, and your code will fail.

To import the driver, type the following command in your shell:

```
>>> import pymongo
```

Once the driver has been loaded and is known to the Python shell, you can start loading the module you want to work with. The `Connection` module enables you to establish connections. Type the following statement in the shell to load the `Connection` module:

```
>>> from pymongo import Connection
```

Once your MongoDB service is up and running (this is mandatory if you wish to connect), then you can go ahead and establish a connection to the service by calling the `Connection` function.

If no additional parameters are given, then the function assumes you want to connect to the service on the localhost (the default port number for the localhost is 27017). The following line establishes the connection:

```
>>> c = Connection()
```

You can see the connection coming in through the MongoDB service shell. Once you establish a connection, you can use the `c` dictionary to refer to the connection, just as you did in the shell with `db` and in PHP with `$c`. Next, select the database that you want to work with, storing that database under the `db` dictionary. You can do this just as you would in the MongoDB shell—in this example, you use the `inventory` database:

```
>>> db = c.inventory
>>> db
Database(Connection('localhost', 27017), u'inventory')
```

The output in the preceding example shows that you that you are connected to the localhost and that you are using the `inventory` database.

Now that the database has been selected, you can select your MongoDB collection in the exact same way. Because you've already stored the database name under the `db` dictionary, you can use that to select the collection's name, which is called `items` in this case:

```
>>> collection = db.items
```

Inserting Data

All that remains is to define the document by storing it in a dictionary. Let's take the preceding example and insert that into the shell:

```
>>> item = {
...     "Type" : "Laptop",
...     "ItemNumber" : "1234EXD",
...     "Status" : "In use",
...     "Location" : {
...             "Department" : "Development",
...             "Building" : "2B",
...             "Floor" : 12,
...             "Desk" : 120101,
...             "Owner" : "Anderson, Thomas"
...     },
...     "Tags" : ["Laptop","Development","In Use"]
... }
```

Once you define the document, you can insert it using the same `insert` function that is available in the MongoDB shell:

```
>>> collection.insert(item)
ObjectId('4c57207b4abffe0e0c000000')
```

That's all there is to it: you define the document and insert it using the `insert` function.

There's one more interesting trick you can take advantage of when inserting documents: inserting multiple documents at the same time. You can do this by specifying both documents in a single dictionary, and then inserting that document afterwards. The result will return two `Object IDs`; pay careful attention to how the brackets are used in the following example:

```
>>> two = [{
...     "Type" : "Laptop",
...     "ItemNumber" : "2345FDX",
```

```
...     "Status" : "In use",
...     "Location" : {
...             "Department" : "Development",
...             "Building" : "2B",
...             "Floor" : 12,
...             "Desk" : 120102,
...             "Owner" : "Smith, Simon"
...     },
...     "Tags" : ["Laptop","Development","In Use"]
... },
... {
...     "Type" : "Laptop",
...     "ItemNumber" : "3456TFS",
...     "Status" : "In use",
...     "Location" : {
...             "Department" : "Development",
...             "Building" : "2B",
...             "Floor" : 12,
...             "Desk" : 120103,
...             "Owner" : "Walker, Jan"
...     },
...     "Tags" : ["Laptop","Development","In Use"]
... }]
>>> collection.insert(two)
[ObjectId('4c57234c4abffe0e0c000001'), ObjectId('4c57234c4abffe0e0c000002')]
```

Finding Your Data

PyMongo provides two functions for finding your data: `find_one()`, which finds a single document in your collection that matches specified criteria; and `find()`, which can find multiple documents based on the supplied parameters (if you do not specify any parameters, `find()` returns all documents in the collection). Let's look at some examples.

Finding a Single Document

As just mentioned, you use the `find_one()` function to find a single document. The function is similar to the `findOne()` function in the MongoDB shell, so mastering how it works shouldn't present much of a challenge for you. By default, this function will return the first document in your collection if it is executed without any parameters, as in the following example:

```
>>> collection.find_one()
    {
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'1234EXD',
    u'Location':{
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Anderson, Thomas',
```

```
        u'Desk': 120101
        },
    u'_id': ObjectId('4c57207b4abffe0e0c000000'),
    u'Type': u'Laptop'
    }
```

You can specify additional parameters to ensure that the first document returned matches your query. The query parameters need to be written just as they would if you were defining them in the shell; that is, you need to specify a key and its value (or a number of values). For instance, assume you want to find a document for which an `ItemNumber` has the value of `3456TFS`. The following query accomplishes that, returning the output as shown:

```
>>> collection.find_one({"ItemNumber" : "3456TFS"})
{
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'3456TFS',
    u'Location': {
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Walker, Jan',
        u'Desk': 120103
    },
        '_id': ObjectId('4c57234c4abffe0e0c000002'),
    u'Type': u'Laptop'
}
```

If the search criteria are relatively common for a document, you can also specify additional query operators. For example, imagine querying for `{"Department" : "Development"}`, which would return more than one result. We'll look at such an example momentarily; however, first let's determine how to return multiple documents, rather than just one. This may actually be a little different than you suspect.

Finding Multiple Documents

You need to use the `find()` function to return more than a single document. You've probably used this command in MongoDB hundreds of times by this point in the book, so you're probably feeling rather comfortable with it. The concept is the same in Python: you specify the query parameters between the brackets to find the actual information.

Getting the results back to your screen, however, works a little differently. Just as when working with PHP and in the shell, querying for a set of documents will return a cursor instance to you. Unlike when typing in the shell, however, you can't simply type in `db.items.find()` to have all results presented to you. Instead, you need to retrieve all documents using the cursor. The following example shows how to display all documents from the `items` collection (note that you previously defined `collection` to match the collection's name; the results are left out for the sake of clarity):

```
>>> for doc in collection.find():
...             doc
...
```

Pay close attention to the indentation before the word `doc`. If this indentation is not used, then an error message will be displayed stating that an expected indented block didn't occur. It's one of Python's strengths that it uses such an indentation method for block delimiters because this approach keeps the code well ordered. Rest assured, you'll get used to this *Pythonic* coding convention relatively quickly. If

you do happen to forget about the indentation, however, you'll see an error message that looks something like this:

```
File "<stdin>", line 2
    doc
      ^
IndentationError: expected an indented block
```

Next, let's look at how to specify a query operator using the `find()` function. The methods used for this are identical to the ones seen previously in the book:

```
>>> for doc in collection.find({"Location.Owner" : "Walker, Jan"}):
...             doc
...
{
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'3456TFS',
    u'Location': {
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Walker, Jan',
        u'Desk': 120103
    },
    u'_id': ObjectId('4c57234c4abffe0e0c000002'),
    u'Type': u'Laptop'
}
```

Using Dot Notation

Dot notation is used to search for matching elements in an embedded object. The preceding snippet actually shows an example of how to do this. When using this technique, you simply specify the key name for an item within the embedded object to search for it, as in the following example:

```
>>> for doc in collection.find({"Location.Department" : "Development"}):
...             doc
...
```

The preceding example returns any document that has the `Development` department set. When searching for information in a simple array (for instance, the tags applied), you only need to fill in any of the matching tags:

```
>>> for doc in collection.find({"Tags" : "Laptop"}):
...             doc
...
```

Returning Fields

If your documents are relatively large, and you do not want to return all key/value information stored in a document, you can include an additional parameter in the `find()` function to specify that only a

certain set of fields need to be returned. You do this by providing a list of field names after the search criteria.

The following example returns the only current owner's name, the item number, and the object ID (this will always be returned, even if you tell it to not show up):

```
>>> for doc in collection.find({'Status' : 'In use'} , {'ItemNumber' : 'true',
'Location.Owner':'true'}):
...             doc
...
{
        u'ItemNumber': u'1234EXD',
        u'_id': ObjectId('4c57207b4abffe0e0c000000'),
        u'Location': {
                u'Owner': u'Anderson, Thomas'
        }
}
{
        u'ItemNumber': u'2345FDX',
        u'_id': ObjectId('4c57234c4abffe0e0c000001'),
        u'Location': {
                u'Owner': u'Smith, Simon'
        }
}
{
        u'ItemNumber': u'3456TFS',
        u'_id': ObjectId('4c57234c4abffe0e0c000002'),
        u'Location': {
                u'Owner': u'Walker, Jan'
        }
}
```

I suspect you'll agree this approach to specifying criteria is quite handy.

Simplifying Queries with Sort, Limit, and Skip

The `sort()`, `limit()`, and `skip()` functions will make implementing your queries much easier. Individually, each of these functions has its charms, but combining them makes them even better and more powerful. You can use the `sort()` function to sort the results by a specific key; the `limit()` function to limit the total number of results returned; and the `skip()` function to skip the first *n* number of items found before returning the remainder of the documents that match your query.

Let's look at a set of individual examples, beginning with the `sort()` function. To save some space, the following example includes another parameter to ensure only a few fields are returned:

```
>>> for doc in collection.find ({"Status" : "In use"},
...    {"ItemNumber":"true", "Location.Owner" : "True"})
...   .sort("ItemNumber"):
...             doc
...
{
        u'ItemNumber': u'1234EXD',
        u'_id': ObjectId('4c57207b4abffe0e0c000000'),
        u'Location': {
                u'Owner': u'Anderson, Thomas'
```

```
        }
}
{
        u'ItemNumber': u'2345FDX',
        u'_id': ObjectId('4c57234c4abffe0e0c000001'),
        u'Location': {
                u'Owner': u'Smith, Simon'
        }
}
{
        u'ItemNumber': u'3456TFS',
        u'_id': ObjectId('4c57234c4abffe0e0c000002'),
        u'Location': {
                u'Owner': u'Walker, Jan'
        }
}
```

Next, let's look at the `limit()` function in action. In this case, you tell the function to return only the `ItemNumber` from the first two items it finds in the collection (note that no search criteria are specified in this example):

```
>>> for doc in collection.find({}, {"ItemNumber" : "true"}).limit(2):
...             doc
...
{u'ItemNumber': u'1234EXD', u'_id': ObjectId('4c57207b4abffe0e0c000000')}
{u'ItemNumber': u'2345FDX', u'_id': ObjectId('4c57234c4abffe0e0c000001')}
```

You can use the `skip()` function to skip a few items before returning a set of documents, as in the following example:

```
>>> for doc in collection.find({}, {"ItemNumber" : "true"}).skip(2):
...     doc
...
{u'ItemNumber': u'3456TFS', u'_id': ObjectId('4c57234c4abffe0e0c000002')}
```

You can also combine the three functions to select only a certain amount of items found, while simultaneously specifying a specific number of items to skip and sorting them:

```
>>> for doc in collection.find( {'Status' : 'In use'},
...     {'ItemNumber':'true', 'Location.Owner':'true'} )
...     .limit(2).skip(1).sort("ItemNumber"):
...             doc
...
{
        u'ItemNumber': u'2345FDX',
        u'_id': ObjectId('4c57234c4abffe0e0c000001'),
        u'Location': {
                u'Owner': u'Smith, Simon'
                }
```

```
}
{
        u'ItemNumber': u'3456TFS',
        u'_id': ObjectId('4c57234c4abffe0e0c000002'),
        u'Location': {
                u'Owner': u'Walker, Jan'
        }
}
```

What you just did—limiting the results returned and skipping a certain number of items—is generally known as *paging*. You can accomplish this in a slightly more simplistic way with the `$slice` operator, which will be covered later in this chapter.

Aggregating Queries

As previously noted, MongoDB comes with a powerful set of aggregation tools (see Chapter 4 for more information on these tools). The cool part: you can use all these tools with the Python driver. These tools make it possible to using the `count()` function to perform a count on your data; using the `distinct()` function to get a list of distinct values with no duplicates; and, last but not least, use the `map_reduce()` function to group your data and batch manipulate the results or simply to perform counts.

This set of commands, used separately or together, enables you to effectively query for the information you need to know—and nothing else.

Counting Items with Count()

You can use the `count()` function if all you want is to perform a count on the total number of items matching your criteria,. The function doesn't return all the information the way the `find()` function does; instead, it returns an integer value with the total of items found.

Let's have a look at some simple examples. Let's begin by returning the total number of documents in the entire collection, without specifying any criteria.

```
>>> collection.find({}).count()
3
```

You can also specify these count queries more precisely, as in this example:

```
>>> collection.find({"Status" : "In use", "Location.Owner" : "Walker, Jan"}).count()
1
```

The `count()` function can be great when all you need is a quick count of the total number of documents that match your criteria.

Counting Unique Items with Distinct()

The `count()` function is a great way to get the total number of items returned. However, sometimes you might accidentally add duplicates to your collection because you simply forget to remove or change an old document, and you want to get an accurate count that shows no duplicates. This is where the `distinct()` function can help you out. This function ensures that only unique items will be returned. Let's set up an example by adding another item to the collection with an `ItemNumber` used previously:

```
>>> dup = ( {
    "ItemNumber" : "2345FDX",
```

```
    "Status" : "Not used",
    "Type" : "Laptop",
    "Location" : {
        "Department" : "Storage",
        "Building" : "1A"
    },
    "Tags" : ["Not used","Laptop","Storage"]
} )
>>> collection.insert(dup)
ObjectId('4c592eb84abffe0e0c000004')
```

When you use the `count()` function at this juncture, the number of *unique* items won't be correct:

```
>>> collection.find({}).count()
4
```

Instead, you can use the `distinct()` function to ensure that any duplicates get ignored:

```
>>> collection.distinct("ItemNumber")
[u'1234EXD', u'2345FDX', u'3456TFS']
```

Grouping Data with map_reduce()

The map_reduce() function is great for grouping your data by a certain tag and performing counts or other type of manipulation on it. The `map_reduce()` function is called from the `Code` module; therefore it needs to be invoked first. Let's have a look at a practical example that counts the occurrence of each tag and returns the results.

Begin by loading the `Code` module; you do this the same way you loaded the `Connection` module at the beginning of this chapter:

```
>>> from pymongo.code import Code
```

Now you're ready to define the `map` dictionary itself. You tell this object to format the data it finds as a set of keys and values. The following example uses the `Tags` key, setting the value of it to `1` because you want every item to count as just one object (this helps keep things simple). Naturally, you can change this number if you want to, but let's leave it as-is for now. The following snippet defines the `map` dictionary:

```
>>> map = Code("function() {"
...     "this.Tags.forEach(function(t) {"
...     "       emit(t, 1);"
...     "});"
...     "}")
```

You've defined the `map` dictionary; next, you need to specify the `reduce` dictionary, so you can accomplish the actual grouping. This requires an initial counter called `Total`, which you set to `0`. This is your default value. If you want to start off with 20 tags each, then you can change this value from `0` to `20`.

The following code specifies the `reduce` dictionary:

```
>>> reduce = Code("function (key, values) {"
...     "       var Total = 0;"
...     "       for (var i = 0; i < values.length; i++) {"
...     "               Total += values[i];"
...     "       }"
...     "       return Total;"
...     "}")
```

Now that the map and reduce dictionaries have been defined, you can go ahead and invoke the map_reduce() command, specifying map and reduce as its parameters. The results will be returned in a cursor, which you will need to treat like any other cursor to return the contents, as in the following example:

```
>>> result = collection.map_reduce(map, reduce)
>>> for tag in result.find():
...             tag
...
{u'_id': u'Development', u'value': 3.0}
{u'_id': u'In Use', u'value': 3.0}
{u'_id': u'Laptop', u'value': 4.0}
{u'_id': u'Not used', u'value': 1.0}
{u'_id': u'Storage', u'value': 1.0}
```

You can also use additional parameters as desired. For instance, you can use the out parameter to define the output collection; the query parameter to define your query; or the limit parameter to limit the total number of results returned. The next example applies these parameters. You don't need to redefine the map or reduce dictionaries this time, so you can skip ahead to executing the map_reduce command itself:

```
>>> result = collection.map_reduce(map, reduce, query={"Status" : "In use"}, limit=4,
out="Tags")
>>> for tag in result.find():
...             tag
...
{u'_id': u'Development', u'value': 3.0}
{u'_id': u'In Use', u'value': 3.0}
{u'_id': u'Laptop', u'value': 3.0}
```

Specifying an Index with Hint()

You can use the hint() function to specify which index ought to be used when querying for data. Using this function helps you to increase the query's performance. In Python, the hint() function also executes on the cursor. However, you should keep in mind that the hint name you specify in Python needs to be the same as the one you passed to the create_index() function.

In the next example, you will create an index first, and then search for the data that specifies the index. Before you can sort in ascending order, however, you will need to use the import() function to import the ASCENDING method. Finally, you need to execute the create_index() function:

```
>>> from pymongo import ASCENDING
>>> collection.create_index([("ItemNumber", ASCENDING)])
u'ItemNumber_1'

>>> for doc in collection.find({"Location.Owner" : "Walker, Jan"}) .hint([("ItemNumber",
ASCENDING)]):
...             doc
...
{
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'3456TFS',
    u'Location': {
```

```
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Walker, Jan',
        u'Desk': 120103
    },
    u'_id': ObjectId('4c57234c4abffe0e0c000002'),
    u'Type': u'Laptop'
}
```

Using indexes can help you significantly increase performance when the size of your collections keeps growing (see Chapter 10 for more details on performance tuning).

Refining Queries with Conditional Operators

You can use conditional operators to refine your query. Python includes more than a half dozen conditional operators; these are identical to the conditional operators you've seen in the previous chapters. The following sections walk you through the conditional operators available in Python, as well as how you can use them to refine your queries in Python.

Using the $lt, $gt, $lte, and $gte Operators

Let's begin by looking at the `$lt`, `$gt`, `$lte`, and `$gte` conditional operators. You can use the `$lt` operator to search for any numerical information that is *less than n*. The operator only takes one parameter: the number `n`, which specifies the limit. The following example finds any entries that have a desk number lower than 120102. Note that the `n` parameter itself is *not* included:

```
>>> for doc in collection.find({"Location.Desk" : {"$lt" : 120102} }):
...             doc
...
{
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'1234EXD',
    u'Location': {
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Anderson, Thomas',
        u'Desk': 120101
    },
    u'_id': ObjectId('4c57207b4abffe0e0c000000'),
    u'Type': u'Laptop'
}
```

In a similar vein, you can use the `$gt` operator to find any items with a value *higher than n*. Again, note that the `n` parameter itself is not included:

```
>>> for doc in collection.find({"Location.Desk" : {"$gt" : 120102} }):
...             doc
...
{
```

```
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'3456TFS',
    u'Location': {
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Walker, Jan',
        u'Desk': 120103
    },
    u'_id': ObjectId('4c57234c4abffe0e0c000002'),
    u'Type': u'Laptop'
}
```

If you want to include the value of the `n` parameters in your results, then you can use either the `$lte` or `$gte` operators to find any values *less than or equal to n* or *greater than or equal to n*, respectively. The following examples illustrate how to use these operators:

```
>>> for doc in collection.find({"Location.Desk" : {"$lte" : 120102} }):
...             doc
...
{
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'1234EXD',
    u'Location': {
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Anderson, Thomas',
        u'Desk': 120101
    },
    u'_id': ObjectId('4c57207b4abffe0e0c000000'),
    u'Type': u'Laptop'
}
{
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'2345FDX',
    u'Location': {
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Smith, Simon',
        u'Desk': 120102
    },
    u'_id': ObjectId('4c57234c4abffe0e0c000001'),
    u'Type': u'Laptop'
}

>>> for doc in collection.find({"Location.Desk" : {"$gte" : 120102} }):
...             doc
...
{
```

```
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'2345FDX',
    u'Location': {
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Smith, Simon',
        u'Desk': 120102
    },
    u'_id': ObjectId('4c57234c4abffe0e0c000001'),
    u'Type': u'Laptop'
}
{
    u'Status': u'In use',
    u'Tags': [u'Laptop', u'Development', u'In Use'],
    u'ItemNumber': u'3456TFS',
    u'Location': {
        u'Department': u'Development',
        u'Building': u'2B',
        u'Floor': 12,
        u'Owner': u'Walker, Jan',
        u'Desk': 120103
    },
    u'_id': ObjectId('4c57234c4abffe0e0c000002'),
    u'Type': u'Laptop'
}
```

Searching for Non-Matching Values with $ne

You can use the $ne (not equals) operator to search for any documents in a collection that do *not* match specified criteria. This operator requires one parameter, the key and value information that a document should not have for the result to return a match:

```
>>> collection.find({"Status" : {"$ne" : "In use"}}).count()
1
```

Specifying an Array of Matches with $in

The $in operator lets you specify an array of possible matches; the SQL equivalent of this operator is IN.

For instance, assume you're looking for only two different kinds of development computers: not used or with Development. Also assume that you want to limit the results to two items, returning only the ItemNumber:

```
>>> for doc in collection.find({"Tags" : {"$in" : ["Not used","Development"]}} ,
{"ItemNumber":"true"}).limit(2):
...             doc
...
{u'ItemNumber': u'1234EXD', u'_id': ObjectId('4c57207b4abffe0e0c000000')}
{u'ItemNumber': u'2345FDX', u'_id': ObjectId('4c57234c4abffe0e0c000001')}
```

Specifying Against an Array of Matches with $nin

You use the `$nin` operator exactly as you use the `$in` operator; the difference is that this operator *excludes* any documents that match any of the values specified in the given array. For example, the following query finds any items that are currently *not* used in the `Development` department:

```
>>> for doc in collection.find({"Tags" : {"$nin" : ["Development"]}}, {"ItemNumber":"true"}):
...             doc
...
{u'ItemNumber': u'2345FDX', u'_id': ObjectId('4c592eb84abffe0e0c000004')}
```

Finding Documents that Match an Array's Values

Whereas the `$in` operator can be used to find any document that matches *any* of the values specified in an array, the `$all` operator lets you find any document that matches *all* of the values specified in an array. The syntax to accomplish this looks exactly the same:

```
>>> for doc in collection.find({"Tags" : {"$all" : ["Storage","Not used"]}},
{"ItemNumber":"true"}):
...             doc
...
{u'ItemNumber': u'2345FDX', u'_id': ObjectId('4c592eb84abffe0e0c000004')}
```

Specifying Multiple Expressions to Match with $or

You can use the `$or` operator to specify multiple values that a document can have to qualify as a match. This is similar to the `$in` operator; the difference is that the `$or` operator lets you specify the key, as well as the value. You can also combine the `$or` operator with another key/value combination. Let's look at a few examples.

This example returns all documents that either have the location set to `Storage` or have the owner set to `Anderson, Thomas`:

```
>>> for doc in collection.find({"$or" : [ { "Location.Department" : "Storage" },
...     { "Location.Owner" : "Anderson, Thomas"} ] } ):
...             doc
...
```

You can also combine the preceding code with another key/value pair, as in this example:

```
>>> for doc in collection.find({ "Location.Building" : "2B", "$or" : [ { "Location.Department"
: "Storage" },
...    { "Location.Owner" : "Anderson, Thomas"} ] } ):
...             doc
...
```

The `$or` operator basically allows you to conduct two searches simultaneously and combine the resulting output, even if the individual searches have nothing in common with each other.

Retrieving Items from an Array with $slice

You can use the `$slice` operator to retrieve a certain number of items from a given array in your document. This operator provides functionality similar to the `skip()` and `limit()` functions; the difference is that those two functions work on full documents, whereas the `$slice` operator works on an array in a *single* document.

Before looking at an example, let's add a new document that will enable us to take a better look at this operator. Assume that your company is maniacally obsessed with tracking its chair inventory, tracking chairs wherever they might go. Naturally, every chair has its own history of desks to which it once belonged. The `$slice` example operator is great for tracking that kind of inventory.

Begin by adding the following document:

```
>>> chair =  ({
...     "Status" : "Not used",
...     "Tags" : ["Chair","Not used","Storage"],
...     "ItemNumber" : "6789SID",
...     "Location" : {
...     "Department" : "Storage",
...     "Building" : "2B"
...     },
...     "PreviousLocation" :
...     [ "120100","120101","120102","120103","120104","120105",
...     "120106","120107","120108","120109","120110" ]
...     })

>>> collection.insert(chair)
ObjectId('4c5973554abffe0e0c000005')
```

Now assume you want to see all the information available for the chair returned in the preceding example, with one caveat: you don't want to see all the previous location information, but only the first three desks it belonged to:

```
>>> collection.find_one({"ItemNumber" : "6789SID"}, {"PreviousLocation" : {"$slice" : 3} })
{
    u'Status': u'Not used',
    u'PreviousLocation': [u'120100', u'120101', u'120102'],
    u'Tags': [u'Chair', u'Not used', u'Storage'],
    u'ItemNumber': u'6789SID',
    u'Location': {
        u'Department': u'Storage',
        u'Building': u'2B'
    },
    u'_id': ObjectId('4c5973554abffe0e0c000005')
}
```

Similarly, you can see its three most recent locations by making the integer value negative:

```
>>> collection.find_one({"ItemNumber" : "6789SID"}, {"PreviousLocation" : {"$slice" : -3} })
{
    u'Status': u'Not used',
    u'PreviousLocation': [u'120108', u'120109', u'120110'],
    u'Tags': [u'Chair', u'Not used', u'Storage'],
    u'ItemNumber': u'6789SID',
    u'Location': {
```

```
        u'Department': u'Storage',
        u'Building': u'2B'
    },
    u'_id': ObjectId('4c5973554abffe0e0c000005')
}
```

Or, you could skip the first five locations for the chair and limit the number of results returned to three (pay special attention to the brackets, here):

```
>>> collection.find_one({"ItemNumber" : "6789SID"}, {"PreviousLocation" : {"$slice" : [5, 3] }
})
{
    u'Status': u'Not used',
    u'PreviousLocation': [u'120105', u'120106', u'120107'],
    u'Tags': [u'Chair', u'Not used', u'Storage'],
    u'ItemNumber': u'6789SID',
    u'Location': {
        u'Department': u'Storage',
        u'Building': u'2B'
    },
    u'_id': ObjectId('4c5973554abffe0e0c000005')
}
```

You probably get the idea. The preceding example might seem a tad unusual, but inventory control systems often veer into the unorthodox; and the `$slice` operator is intrinsically good at helping you account for unusual or complex circumstances. For example, the `$slice` operator might prove an especially effective tool for implementing the paging system for a website's Comments section, as you see in the next chapter.

Conducting Searches with Regular Expression

One useful tool for conducting searches is Regular Expression. The default Regular Expression module for Python is called `re`. Performing a search with the `re` module requires that you first load the module, as in this example:

```
>>> import re
```

After you load the module, you can specify the Regular Expression query in the `value` field of your search criteria. The following example shows how to search for any document where `ItemNumber` has a value that contains a `4` (for the sake of keeping things simple, this example returns only the values in `ItemNumber`):

```
>>> for doc in collection.find({"ItemNumber" : re.compile("4")}, {"ItemNumber" : "true"}):
...             doc
...
{u'ItemNumber': u'1234EXD', u'_id': ObjectId('4c57207b4abffe0e0c000000')}
{u'ItemNumber': u'2345FDX', u'_id': ObjectId('4c57234c4abffe0e0c000001')}
{u'ItemNumber': u'2345FDX', u'_id': ObjectId('4c592eb84abffe0e0c000004')}
{u'ItemNumber': u'3456TFS', u'_id': ObjectId('4c57234c4abffe0e0c000002')}
```

You can further define a Regular Expression. At this stage, your query is case sensitive, and it will match any document that has a `4` in the value of `ItemNumber`, regardless of its position. However, assume you want to find a document where the value of `ItemNumber` ends with `FS`, is preceded by an unknown value, and can contain no additional data after the `FS`:

```
>>> for doc in collection.find({"ItemNumber" : re.compile(".FS$")}, {"ItemNumber" : "true"}):
...             doc
...
{u'ItemNumber': u'3456TFS', u'_id': ObjectId('4c57234c4abffe0e0c000002')}
```

You can also to search for information in a case-insensitive way, but first you must add another function, as in this example:

```
>>> for doc in collection.find({"Location.Owner" : re.compile("^anderson.", re.IGNORECASE)},
...     {"ItemNumber" : "true", "Location.Owner" : "true"}):
...             doc
...
{
     u'ItemNumber': u'1234EXD',
     u'_id': ObjectId('4c57207b4abffe0e0c000000'),
     u'Location': {
          u'Owner': u'Anderson, Thomas'
     }
}
```

Regular Expression can be an extremely powerful tool, as long as you utilize it properly. For more details on how the `re` module works and which functions it includes, please refer to the module's official documentation at `http://docs.python.org/library/re.html`.

Modifying the Data

So far you've learned how to use conditional operators and Regular Expression in Python to query for information in your database. In the next part of this chapter, we'll examine how to use Python to modify the existing data in your collections. We can use Python to accomplish this task in several different ways. The upcoming sections will build on the previously used query operators to find the documents that ought to match your modifications. In a couple cases, you may need to skip back to earlier parts of this chapter to brush up on particular aspects of using query operators—but that's a normal part of the learning process, and it will reinforce the lessons taught so far.

Updating Your Data

The way you use Python's `update()` function doesn't vary much from how you use the identically named function in the MongoDB shell or the PHP driver. In this case, you provide two mandatory parameters to update your data: `arg` and `doc`. The `arg` parameter specifies the key/value information used to match a document, while the `doc` parameter contains the updated information. You can also specify four optional parameters. The following list covers Python's list of parameters to update information, including what they do:

- `arg`: Specifies the search arguments (key/value information) that a document must contain to qualify for the update. These arguments can be either a dictionary or a set of key/value information that is stored in a SON object.
- `doc`: Comprises either a *dictionary* or a *SON* object that contains the information to update the matching document with.
- `upsert` *(optional)*: If set to `true`, performs an upsert.

- `manipulate` *(optional)*: If set to `true`, indicates the document will be manipulated *before* performing the update using all instances of the *SONManipulator*.
- `safe` *(optional)*: If set to `true`, performs a check to see whether the update succeeded.
- `multi` *(optional)*: If set to `true`, updates any matching document, rather than just the first document it finds (the default action). It is recommended that you always set this to `true` or `false`, rather than relying on the default behavior (which could always change in the future).

If you do not specify any of the modifier operators when updating a document, then by default all information in the document will be replaced with whatever data you inserted in the `doc` parameter. It is best to avoid relying on the default behavior; instead, you should use the aforementioned operators to specify your desired updates explicitly (you'll learn how to do this momentarily).

You can see why it's best to use conditional operators with the `update()` command by looking at a case where you don't use any conditional operators with the command:

```
// Define the updated data
>>> update = ( {
    "Type" : "Chair",
    "Status" : "In use",
    "Tags" : ["Chair","In use","Marketing"],
    "ItemNumber" : "6789SID",
    "Location" : {
        "Department" : "Marketing",
        "Building" : "2B",
        "DeskNumber" : 131131,
        "Owner" : "Martin, Lisa"
    }
} )

// Now, perform the update
>>> collection.update({"ItemNumber" : "6789SID"}, update)

// Confirm the update was successful
>>> collection.find_one({"Type" : "Chair"})
{
    u'Status': u'In use',
    u'Tags': [u'Chair', u'In use', u'Marketing'],
    u'ItemNumber': u'6789SID',
    u'Location': {
        u'Department': u'Marketing',
        u'Building': u'2B',
        u'DeskNumber': 131131,
        u'Owner': u'Martin, Lisa'
    },
    u'_id': ObjectId('4c5973554abffe0e0c000005'),
    u'Type': u'Chair'
}
```

One big minus about the preceding example: it's somewhat lengthy, and it updates only a few fields. Next, we'll look at what the modifier operators can be used to accomplish.

Modifier Operators

Chapter 4 detailed how the MongoDB shell includes a large set of modifier operators that you can use to manipulate your data more easily, but without needing to rewrite the entire document to change a single field's value (as seen in the preceding example).

The modifier operators let you do everything from changing one existing value in a document, to inserting an entire array, to removing all entries from multiple items specified in an array. As a group, these operators make it easy to modify data. Now let's take a look at what the operators do and how you use them.

Increasing an Integer Value with $inc

You use the `$inc` operator to increase an integer value in a document by the given number, *n*. The following example shows how to increase the integer value of `Location.Desknumber` by 20:

```
>>> collection.update({"ItemNumber" : "6789SID"}, {"$inc" : {"Location.DeskNumber" : 20}})
```

Next, check to see whether the update was successful:

```
>>> collection.find_one({"Type" : "Chair"}, {"Location" : "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Location': {
          u'Department': u'Marketing',
          u'Building': u'2B',
          u'Owner': u'Martin, Lisa',
          u'DeskNumber': 131151
     }
}
```

Note that the `$inc` operator only works on integer values (i.e., numeric values), but not on any string values or even numeric values added as a string (e.g., `"123"` vs. `123`).

Changing an Existing Value with $set

You use the `$set` operator to change an existing value in any matching document. This is an operator you'll use frequently. The next example changes the value from `"Building"` in any item currently matching the key/value `"Location.Department / Development"`.

You use `$set` to perform the update, ensuring that all documents are updated and all upserts are performed:

```
>>> collection.update({"Location.Department" : "Development"},
...    {"$set" : {"Location.Building" : "3B"} },
...    upsert = True, multi = True )
```

Next, use the `find_one()` command to confirm all went well:

```
>>> collection.find_one({"Location.Department" : "Development"}, {"Location.Building" : True})
{
     u'_id': ObjectId('4c57207b4abffe0e0c000000'),
     u'Location': {u'Building': u'3B'}
}
```

Removing a Key/Value Field with $unset

Likewise, you use the `$unset` operator to remove a key/value field from a document, as shown in the following example:

```
>>> collection.update({"Status" : "Not used", "ItemNumber" : "2345FDX"},
...     {"$unset" : {"Location.Building" : 1 } } )
```

Next, use the `find_one()` command to confirm all went well:

```
>>> collection.find_one({"Status" : "Not used", "ItemNumber" : "2345FDX"}, {"Location" :
"True"})
{
     u'_id': ObjectId('4c592eb84abffe0e0c000004'),
     u'Location': {u'Department': u'Storage'}
}
```

Adding a Value to an Array with $push

The `$push` operator lets you add a value to an array, assuming the array exists. If the array does not exist, then it will be created with the value specified.

■ **Warning** If you use `$push` to update an existing field that isn't an array, an error message will pop up.

Now you're ready to add a value to an already existing array and confirm whether all went well. First, perform the update:

```
>>> collection.update({"Location.Owner" : "Anderson, Thomas"},
...    {"$push" : {"Tags" : "Anderson"} }, multi = True )
```

Now, execute `find_one()` to confirm whether the update(s) went well:

```
>>> collection.find_one({"Location.Owner" : "Anderson, Thomas"}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c57207b4abffe0e0c000000'),
     u'Tags': [u'Laptop', u'Development', u'In Use', u'Anderson']
}
```

Adding Multiple Values to an Array with $pushAll

The `$pushAll` operator is similar to the `$push` operator, with one essential difference: it lets you add multiple values to an existing array. Again, the array must already exist, or you will receive an error. The following example uses $pushAll in conjunction with Regular Expression to perform a search; this enables you to apply a change to all matching queries:

```
>>> collection.update({"Location.Owner" : re.compile("^Walker,")},
...    {"$pushAll" : {"Tags" : ["Walker","Warranty"] } } )
```

Next, execute `find_one()` to see whether all went well:

```
>>> collection.find_one({"Location.Owner" : re.compile("^Walker,")}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c57234c4abffe0e0c000002'),
     u'Tags': [u'Laptop', u'Development', u'In Use', u'Walker', u'Warranty']
}
```

Adding a Value to an Existing Array with $addToSet

The `$addToSet` operator also lets you add a value to an already existing array. The difference is that this method checks whether the array already exists before attempting the update (the `$push` and `$pushAll` operators do not check for this condition).

This operator only takes one additional value; however, it's also good to know that you can combine this operator with the `$each` operator. Let's look at two examples. First, let's perform the update using the `$addToSet` operator on any object matching `"Type : Chair"` and then check whether all went well using the `find_one()` function:

```
>>> collection.update({"Type" : "Chair"}, {"$addToSet" : {"Tags" : "Warranty"} }, multi =
True)

>>> collection.find_one({"Type" : "Chair"}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Tags': [u'Chair', u'In use', u'Marketing', u'Warranty']
}
```

You can also use the `$each` statement to add multiple tags. Note that you perform this search using a Regular Expression. Also, one of the tags in the list has been previously added; fortunately, it won't be added again because this is what `$addToSet` specifically prevents:

```
// Use the $each operator to add multiple tags, including one that was already added
>>> collection.update({"Type" : "Chair", "Location.Owner" : re.compile("^Martin,")},
...    {"$addToSet" : { "Tags" : {"$each" : ["Martin","Warranty","Chair","In use"] } } } )
```

Now it's time to check whether all went well; specifically, you want to verify that the duplicate `Warranty` tag has not been added again:

```
>>> collection.find_one({"Type" : "Chair", "Location.Owner" : re.compile("^Martin,")}, {"Tags"
: "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Tags': [u'Chair', u'In use', u'Marketing', u'Warranty', u'Martin']
}
```

Removing an Element from an Array with $pop

So far, you've seen how to use the update() function to add values to an existing document. Now let's turn this around and look at how to remove data instead. We'll begin by look into the $pop operator.

This operator allows you to delete either the first or last value from an array, but nothing in between. The following example removes the first value in the Tags array from the first document it finds that matches the "Type" : "Chair" criteria; the example then uses the find_one() command to confirm all went well with the update:

```
>>> collection.update({"Type" : "Chair"}, {"$pop" : {"Tags" : -1}})

>>> collection.find_one({"Type" : "Chair"}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Tags': [u'In use', u'Marketing', u'Warranty', u'Martin']
}
```

Giving the Tags array a positive value instead removes the last occurrence in an array, as in the following example:

```
>>> collection.update({"Type" : "Chair"}, {"$pop" : {"Tags" :  1}})
```

Next, execute the find_one() function again to confirm that all went well:

```
>>> collection.find_one({"Type" : "Chair"}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Tags': [u'In use', u'Marketing', u'Warranty']
}
```

Removing a Specific Value with $pull

The $pull operator lets you remove each occurrence of a specific value from an array, regardless of how many times the value occurs; as long as the value is the same, it will be removed.

Let's look at an example. Begin by using the $push operator to add identical tags with the value Double to the Tags array:

```
>>> collection.update({"Type" : "Chair"}, {"$push" : {"Tags" : "Double"} }, multi = False )
>>> collection.update({"Type" : "Chair"}, {"$push" : {"Tags" : "Double"} }, multi = False )
```

Next, ensure that the tag was added twice by executing the find_one() command. Once you confirm the tag exists twice, use the $pull operator to remove both instances of the tag:

```
>>> collection.find_one({"Type" : "Chair"}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Tags': [u'In use', u'Marketing', u'Warranty', u'Double', u'Double']
}

>>> collection.update({"Type" : "Chair"}, {"$pull" : {"Tags" : "Double"} }, multi = False)
```

To confirm all went well, execute find_one() command again, this time making sure that the result no longer lists Double tag:

```
>>> collection.find_one({"Type" : "Chair"}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Tags': [u'In use', u'Marketing', u'Warranty']
}
```

You can use the `$pullAll` operator to perform the same action; the difference is that the `$pullAll` operator lets you remove multiple tags. Again, let's look at an example. First, you need to add multiple items into the `Tags` array again and confirm that they have been added:

```
>>> collection.update({"Type" : "Chair"}, {"$addToSet" : { "Tags" : {"$each" :
["Bacon","Spam"] } } } )
>>> collection.find_one({"Type" : "Chair"}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Tags': [u'In use', u'Marketing', u'Warranty', u'Bacon', u'Spam']
}
```

Now you can use `$pullAll` operator to remove the multiple tags. The following example shows how to use this operator; the example also executes a `find_one()` command immediately afterward to confirm that the `Bacon` and `Spam` tags have been removed:

```
>>> collection.update({"Type" : "Chair"}, {"$pullAll" : {"Tags" : ["Bacon","Spam"] } }, multi
= False)
>>> collection.find_one({"Type" : "Chair"}, {"Tags" : "True"})
{
     u'_id': ObjectId('4c5973554abffe0e0c000005'),
     u'Tags': [u'In use', u'Marketing', u'Warranty']
}
```

Saving Documents Quickly with Save()

You can use the `save()` function to quickly add a document through the `upsert` method. For this to work, you must also define the value of the `_id` field. If the document you want to save already exists, then the document will be updated; if it does not exist already, then it will be created.

Let's look at an example that saves a dictionary called `Desktop`. Begin by specifying the dictionary by typing it into the shell with an identifier, after which you can save it with the `save()` function. Executing the `save()` function returns the `Object ID` from the document once the save is successful:

```
>>> Desktop = ( {
    "Status" : "In use",
    "Tags" : ["Desktop","In use","Marketing","Warranty"],
    "ItemNumber" : "4532FOO",
    "Location" : {
        "Department" : "Marketing",
        "Building" : "2B",
        "Desknumber" : 131131,
        "Owner" : "Martin, Lisa",
    }
} )
>>> collection.save(Desktop)
ObjectId('4c5ddbe24abffe0f34000001')
```

Now assume you realize that you forgot to specify a key/value pair in the dictionary. You can easily add this information to the dictionary by defining the dictionary's name, followed by its key between brackets, and then including the desired contents. Once you do this, you can perform the upsert by simply saving the entire dictionary again; doing so returns the `Object ID` again from the document:

```
>>> Desktop[ "Type" ] = "Desktop"
>>> collection.save(Desktop)
ObjectId('4c5ddbe24abffe0f34000001')
```

As you can see, the value of the `Object ID` returned is unchanged.

Modifying a Document Atomically

You can use the `findAndModify()` function to modify a document atomically and return the results. The Python driver currently does not have a helper method for this function, however, so it needs to be executed as a database command (see Chapter 4 for more information on atomic updates).

The `findAndModify()` function can be used to update only a single document—and nothing more. You should also keep in mind the fact that the document returned will not include the modifications made by default; getting this information requires that you specify an additional argument.

The `findAndModify()` function can be used with seven parameters, and you must include either the `update` parameter or the `remove` parameter. The following list covers all the available parameters, explaining what they are and what they do:

- `query`: Specifies a filter for the query. If this isn't specified, then all documents in the collection will be seen as possible candidates, after which the first document it encounters will be updated or removed.
- `sort`: Sorts the matching documents in a specified order.
- `remove`: If set to `true`, removes the first matching document.
- `update`: Specifies the information to update the document with. Note that any of the modifying operators specified previously can be used for this.
- `new`: If set to `true`, returns the updated document rather than the *selected* document. This is not set by default, however, which might be a bit confusing sometimes.
- `fields`: Specifies the fields you would like to see returned, rather than the entire document. This works identically to the `find()` function. Note that the `_id` field will always be returned.
- `upsert` *(optional)*: If set to `true`, performs an upsert.

Putting the Parameters to Work

You know what the parameters do; now it's time to use them in a real-world example in conjunction with the `findAndModify()` function. Begin by using the `findAndModify()` function to search for any document that has a key/value pair of `"Type" : "Desktop"`—and then update each document that matches the query by setting an additional key/value pair of `"Status" : "In repair"`. Finally, you want to ensure that the updated document(s) gets returned, rather than the old document(s) matching the query:

```
>>> db.command("findandmodify", "items", query = {"Type" : "Desktop"},
...    update = {"$set" : {"Status" : "In repair"} }, new = True  )
```

```
{
    u'ok': 1.0,
    u'value': {
        u'Status': u'In repair',
        u'Tags': [u'Desktop', u'In use', u'Marketing', u'Warranty'],
        u'ItemNumber': u'4532FOO',
        u'Location': {
            u'Department': u'Marketing',
            u'Building': u'2B',
            u'Owner': u'Martin, Lisa',
            u'Desknumber': 131131
        },
        u'_id': ObjectId('4c5dda114abffe0f34000000'),
        u'Type': u'Desktop'
    }
}
```

Let's look at another example. This time, you will use `findAndModify()` to remove a document; in this case, the output will show which document was removed:

```
>>> db.command("findandmodify", "items", query = {"Type" : "Desktop"},
...    sort = {"ItemNumber" : -1}, remove = True  )
{
    u'ok': 1.0,
    u'value': {
        u'Status': u'In use',
        u'Tags': [u'Desktop', u'In use', u'Marketing', u'Warranty'],
        u'ItemNumber': u'4532FOO',
        u'Location': {
            u'Department': u'Marketing',
            u'Building': u'2B',
            u'Owner': u'Martin, Lisa',
            u'Desknumber': 131131
        },
        u'_id': ObjectId('4c5ddbe24abffe0f34000001'),
        u'Type': u'Desktop'
    }
}
```

Deleting Data

In most cases, you will use the Python driver to add or modify your data. However, it's also important to understand how to *delete* data. The Python driver provides several methods for deleting data. First, you can use the `remove()` function to delete a single document from a collection. Second, you can use the `drop()` or `drop_collection()` function to delete an entire collection. Finally, you can use the `drop_database()` function to drop an entire database (it seems unlikely you'll be using this function frequently!).

Nevertheless, we will take a closer look at each of these functions, looking at examples for all of them.

Let's begin by looking at the `remove()` function. This function allows you to specify an argument as a parameter that will be used to find and delete any matching documents in your current collection. In this example, you use the `remove()` function to remove each document that has a key/value pair of `"Status" : "In use"`; afterward, you use the `find_one` command to confirm the results:

```
>>> collection.remove({"Status" : "In use"})
>>> collection.find_one({"Status" : "In use"})
>>>
```

You need to be careful what kind of criteria you specify with this function. Usually, you should execute a `find()` first, so you can see exactly which documents will get removed. Alternatively, you can use the `Object ID` to remove an item.

If you get tired of an entire collection, you can look into using either the `drop()` or the `drop_collection()` function to remove it. Both functions work the same way (one is just an alias for the other, really); specifically, both expect only one parameter, the collection's name:

```
>>> db.items.drop()
```

Last (and far from least because of its potential destructiveness), the `drop_database()` function enables you to delete an entire database. You call this function using the `Connection` module, as in the following example:

```
>>> c.drop_database("inventory")
```

Creating a Link Between Two Documents

Database references can be used to create a link between two documents that reside in different locations. For example, you might create one collection for all employees and another collection for all the items—and then use the `DBRef()` function to create a reference between the employees and the location of the items, rather than typing them in manually for each item.

As you may recall from the previous chapters, you can reference data in one of two ways. First, you can add a simple reference (*manual referencing*) that uses the `_id` field from one document to store a reference to it in another. Second, you can use the `DBRef` module, which brings a few more options with it than you get with manual referencing.

Let's create a manual reference first. Begin by saving a document. For example, assume you want to save the information for a person into a specific collection. The following example defines a `jan` dictionary and saves it into the `people` collection to get back an `Object ID`:

```
>>> jan = {
...     "First Name" : "Jan",
...     "Last Name" : "Walker",
...     "Display Name" : "Walker, Jan",
...     "Department" : "Development",
...     "Building" : "2B",
...     "Floor" : 12,
...     "Desk" : 120103,
...     "E-Mail" : "jw@example.com"
... }

>>> people = db.people
>>> people.insert(jan)
ObjectId('4c5e5f104abffe0f34000002')
```

After you add an item and get its ID back, you can use this information to link the item to another document in another collection:

```
>>> laptop = {
...     "Type" : "Laptop",
```

```
...     "Status" : "In use",
...     "ItemNumber" : "12345ABC",
...     "Tags" : ["Warranty","In use","Laptop"],
...     "Owner" : jan[ "_id" ]
... }
>>> items = db.items
>>> items.insert(laptop)
ObjectId('4c5e6f6b4abffe0f34000003')
```

Now assume you want to find out the owner's information. In this case, all you have to do is query for the `Object ID` given in the `Owner` field; obviously, this is only possible if you know which collection the data is stored in.

But assume that you don't know where this information is stored. It was for handling precisely such scenarios that the `DBRef()` function was created. You can use this function even when you do not know which collection holds the original data. This fact means you don't have to worry so much about the collection names when searching for the information.

The `DBRef()` function takes three arguments; it can take a fourth argument that you can use to specify additional keyword arguments. Here's a list of the three main arguments and what they let you do:

- `collection` *(mandatory)*: Specifies the collection the original data resides in (e.g., `people`).
- `id` *(mandatory)*: Specifies the `_id` value of the document that should be referred to.
- `database` *(optional)*: Specifies the name of the database to reference.

The `DBRef` module must be loaded before you can use the `DBRef` method, so let's load the module before going any further:

```
>>> from pymongo.dbref import DBRef
```

At this point, you're ready to look at a practical example that leverages the `DBRef()` function. In the following example, you insert a person into the `people` collection and add an item to the `items` collection, using `DBRef` to reference the owner:

```
>>> mike = {
...     "First Name" : "Mike",
...     "Last Name" : "Wazowski",
...     "Display Name" : "Wazowski, Mike",
...     "Department" : "Entertainment",
...     "Building" : "2B",
...     "Floor" : 10,
...     "Desk" : 120789,
...     "E-Mail" : "mw@monsters.inc"
... }

>>> people.save(mike)
ObjectId('4c5e73714abffe0f34000004')
```

At this point, nothing interesting has happened. Yes, you added a document, but you did so without adding a reference to it. However, you do have the `Object ID` of the document, so now you can add your next document to the collection, and then use `DBRef()` to point the `owner` field at the value of the previously inserted document. Pay special attention to the syntax of the `DBRef()` function; in particular, you should note how the first parameter given is the collection name where your previously specified

document resides, while the second parameter is nothing more than a reference to the `_id` key in the `mike` dictionary:

```
>>> laptop = {
...     "Type" : "Laptop",
...     "Status" : "In use",
...     "ItemNumber" : "2345DEF",
...     "Tags" : ["Warranty","In use","Laptop"],
...     "Owner" : DBRef('people', mike[ "_id" ])
... }


>>> items.save(laptop)
ObjectId('4c5e740a4abffe0f34000005')
```

As you probably noticed, this code isn't massively different from the code you used to create a manual reference. However, we recommend that you use the `DBRef` method just in case you need to reference specific information, rather than embedding it. Adopting this approach gives you the additional flexibility of not having to look up the collection's name whenever you query for the referenced information.

Retrieving the Information

You know how to reference information with `DBRef()`; now let's assume that you want to retrieve the previously referenced information. You can accomplish this using the Python driver's `dereference()` function. All you need to do is define the field previously specified that contains the referenced information as an argument, and then press the Return key.

Next, let's walk through the process of referencing and retrieving information from one document to another from start to finish. Let's begin by finding the document that contains the referenced data, and then retrieving that document for display. The first step is to create a query that finds a random document with the reference information in it:

```
>>> items.find_one({"ItemNumber" : "2345DEF"})
{
    u'Status': u'In use',
    u'Tags': [u'Warranty', u'In use', u'Laptop'],
    u'ItemNumber': u'2345DEF',
    u'Owner': DBRef(u'people', ObjectId('4c5e73714abffe0f34000004')),
    u'_id': ObjectId('4c5e740a4abffe0f34000005'),
    u'Type': u'Laptop'
}
```

Next, you want to store this item under a `person` dictionary:

```
>>> person = items.find_one({"ItemNumber" : "2345DEF"})
```

At this point, you can use the `dereference()` function to dereference the `Owner` field to the `person["Owner"]` field as an argument. This is possible because the `Owner` field is linked to the data you want to retrieve:

```
>>> db.dereference(person["Owner"])

{
    u'Building': u'2B',
```

```
    u'Floor': 10,
    u'Last Name': u'Wazowski',
    u'Desk': 120789,
    u'E-Mail': u'mw@monsters.inc',
    u'First Name': u'Mike',
    u'Display Name': u'Wazowski, Mike',
    u'Department': u'Entertainment',
    u'_id': ObjectId('4c5e73714abffe0f34000004')
}
```

That wasn't so bad! The point to take away from this example is that `DBRef` provides a great way for storing data you want to reference. Additionally, `DBRef` permits some flexibility in how you specify the collection and database names. You'll find yourself using this feature frequently if you want to keep your database tidy, especially in cases where the data really shouldn't be embedded.

Summary

In this chapter, we've explored the basics of how MongoDB's Python driver (PyMongo) can be used for the most frequently used operations. Along the way, we've covered how to search for, store, update, and delete data.

We've also looked at how to reference documents contained in another collection using two methods: manual referencing and `DBRef`. When looking at these approaches, we've seen how their syntax is remarkably similar, but the `DBRef` approach provides a bit more robustness in terms of its functionality, so is preferable in most circumstances.

The next chapter will delve into how MongoDB's innate flexibility can be used to leverage the PHP driver to create a simple web application.